High Dynamic Range Rendering with God Rays Effect

Intel® OpenCL SDK Sample Documentation

Document Number: 325263-002US

Legal Information

Contents

About God Rays 5
Path 5
Introduction 6
Motivation 6
Algorithm 6
    Original Algorithm 6
    Optimized Algorithm for CPUs 8
OpenCL™ Implementation 13
    Code Highlights 13
    Work-group Size Considerations 13
Understanding OpenCL Performance Characteristics 14
    Benefits of Using Vector Data Types 14
Limitations 14
Future Work and Enhancements 14
Project Structure 15
APIs Used 15
Reference (Native) Implementation 16
Controlling the Sample 16
References 16

About God Rays

God Rays sample demonstrates how to use high dynamic range (HDR) rendering with God Rays (crepuscular rays) effect in OpenCL™. This implementation optimizes rendering passes by sharing intermediate data between pixels during pixel processing, improves the method performance, and reduces data loads. The following figure illustrates God Rays effect applied to an HDR image:

Path

Location	Executable
`<INSTALL_DIR>samples\GodRays`	`Win32\Release\GodRays.exe` – 32-bit executable `x64\Release\GodRays.exe` – 64-bit executable `Win32\Debug\GodRays.exe` – 32-bit debug executable `x64\Debug\GodRays.exe` – 64-bit debug executable

Introduction

Observing an object, you can sometimes also see small particles in the air, such as dust. These particles cause light scattering in the atmosphere that makes sunlight visible. This effect is called *God Rays*. In real-time rendering, you can usually simulate the light scattering by implementing low-frequency effects in screen space.

To emulate the God Rays effect, this sample implements GPU Gems 3 algorithm [1].

Motivation

A good-quality God Rays effect requires many samples along the ray as well as a significant number of calculations and color buffer reads that are also required for smoothing and blurring post-processing. This sample implementation minimizes color buffer accesses and uses data-level parallelism. This results in significant performance gain and better result quality as compared to applications that use the same post-processing effects optimized for traditional GPU architectures.

This sample demonstrates a CPU-optimized implementation of the God Rays effect, showing how to:

- implement calculation kernels using OpenCL C99
- parallelize the kernels by running several work-groups in parallel
- organize data exchange between the host and the OpenCL device
- store the final image on the hard drive.

Algorithm

Original Algorithm

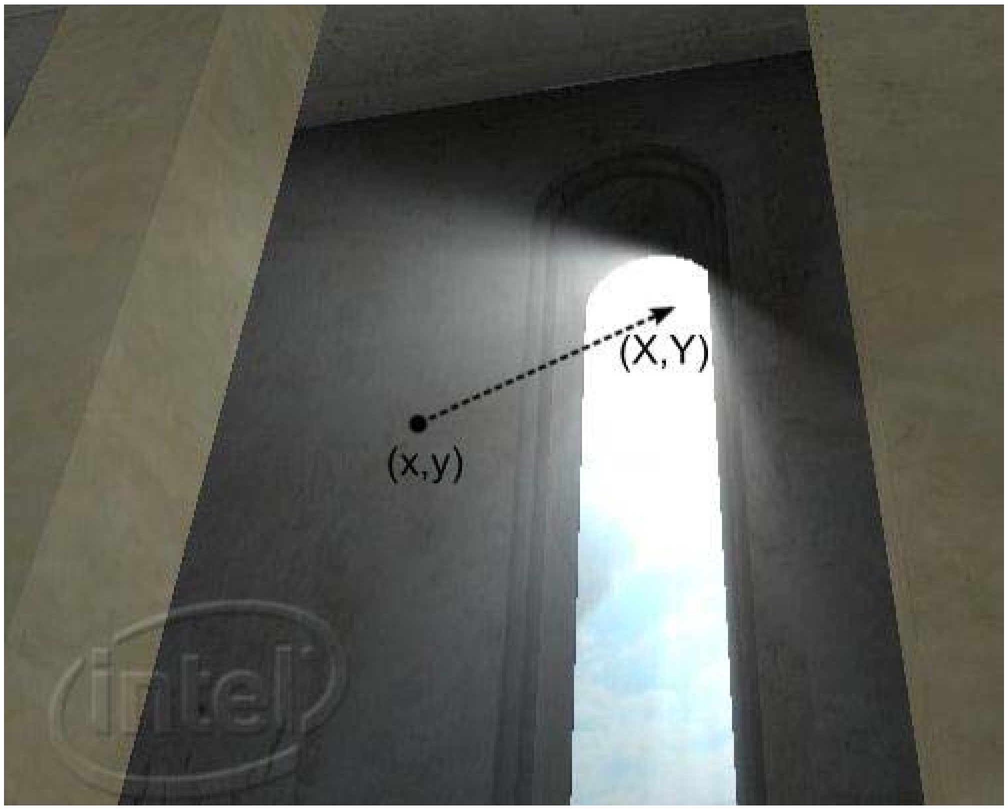
The original algorithm [1] consists of the following stages:

1. For each pixel `(x,y)`, take a segment `[x,y; X,Y]`, where `(X,Y)` is the position of the light source radiating the God Rays.
2. Take N sample points (x_i, y_i) evenly distributed on the segment in the input image space, where $(x_0, y_0) = (x, y)$ and $(x_N, y_N) = (X, Y)$.
3. For each sample point, take HDR values of the source image pixels and sum them up with the *weight* and *decay* coefficients:

$$GodRaysColor = weight * \sum_{i=0}^{N} PixelColor(x_i, y_i) * decay^i$$,

where: *weight* controls intensity of the God Rays effect, $decay_i$ dissipates the contribution along the ray.

The derived sum is the God Rays effect value for pixel (x,y).



`Height*Width*N` is the number of algorithm iterations, where `N` is the number of samples along the ray, `Height` and `Width` are the image height and width, in pixels. For smooth results, set the maximum possible value for `N`. For a small number steps, the sample omits too many pixels in a segment. As a result, the sample computes some neighboring pixels in the ray by different source pixels, and the resulting pixels have different luminosity. The best variant is `N = max(Height, Width)`. Thus, the computational complexity of the algorithm is $O(M^3)$ where `M = max(Height, Width)`.

Optimized Algorithm for CPUs

The optimized algorithm uses God Rays values calculated for pixels on the ray to compute God Rays values for other pixels. Moving from the light source to the image edge along the ray, the optimized implementation calculates the God Rays effect sample values for all affected pixels.

To compute the God Rays effect value for pixel (x_i,y_i), the method takes *i* sample points evenly distributed on the segment [X,Y; x_i,y_i] in the input image space. For each sample point, the method takes HDR values of the source image pixels and sums them up with the weighting and decay coefficients.

$$GodRaysColor_i = weight * \sum_{j=i}^{0} PixelColor(x_j, y_j) * decay^{i-j}$$

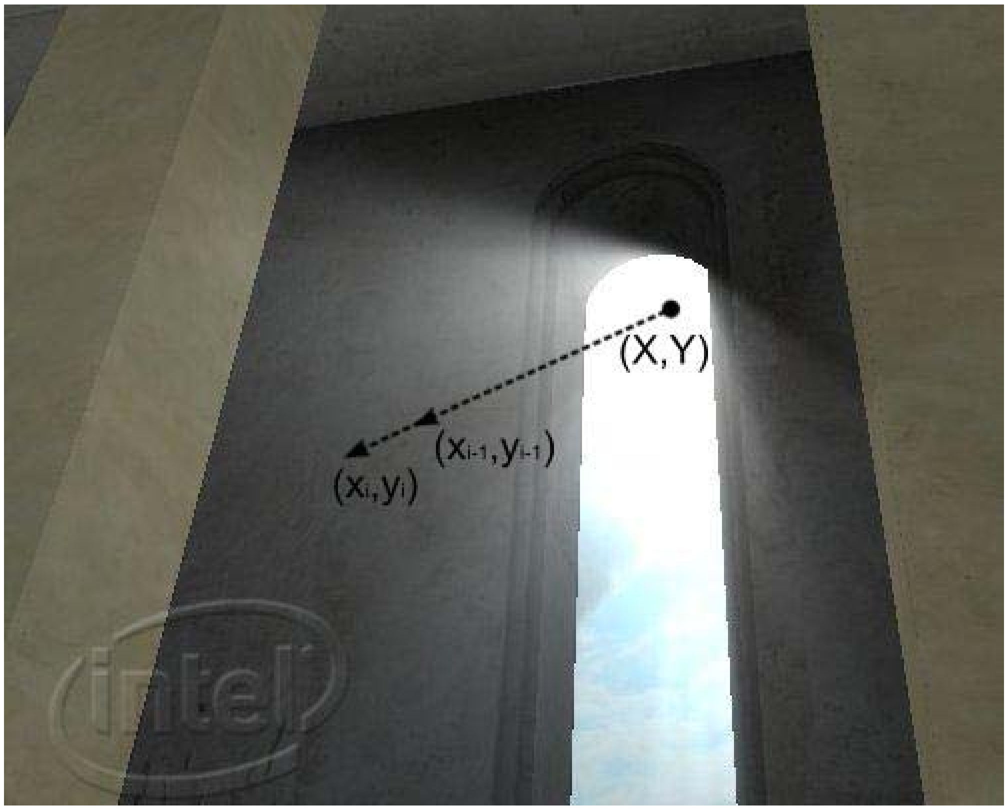
$$GodRaysColor_i = weight * \left(PixelColor(x_i, y_i) + decay * \sum_{j=i-1}^{0} PixelColor(x_j, y_j) * decay^{i-j-1} \right)$$

$$GodRaysColor_i = weight * PixelColor(x_i, y_i) + decay * GodRaysColor_{i-1}$$

$$GodRaysColor_0 = weight * PixelColor(x_0, y_0)$$

The implementation uses the integer Brezenham's line algorithm [3], [4]. At each step, the algorithm takes a single value of the input sample and modifies the accumulated sum.

Use the sum calculated for the pixel (x_{i-1},y_{i-1}) to calculate the sum for the pixel (x_i,y_i) residing on the ray [X,Y; x_i,y_i] immediately after the pixel (x_{i-1},y_{i-1}).

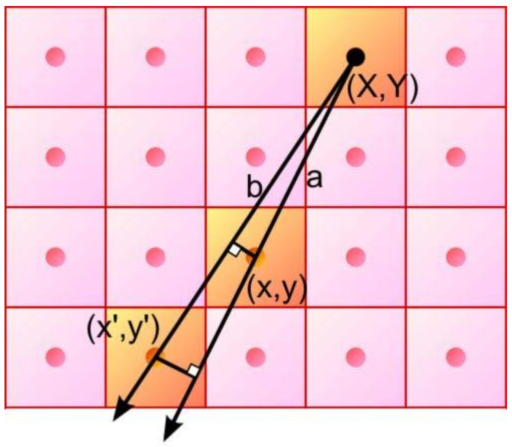


Algorithm Statement

To calculate the God Rays effect values in all pixels of the God Rays mask, you only need to calculate sample values along all the rays from the light source position to each boundary pixel of the frame.

Algorithm Proof

1. For each non-boundary pixel (x,y), take the boundary pixel (x',y'), in which the distance from the center to the ray a = [X,Y; x,y] is minimal. As per Brezenham's line algorithm, this ray a = [X,Y; x,y] intersects the edged pixel.
2. Draw the ray b = [X,Y; x',y'] through the center of the pixel (x',y'). As per the triangle inequality, the distance from ray *b* to the center of the pixel (x,y) is not longer than the distance from the source ray to the center of the edged pixel. Therefore, as per Brezenham's line algorithm, ray *b* cuts the pixel (x,y).

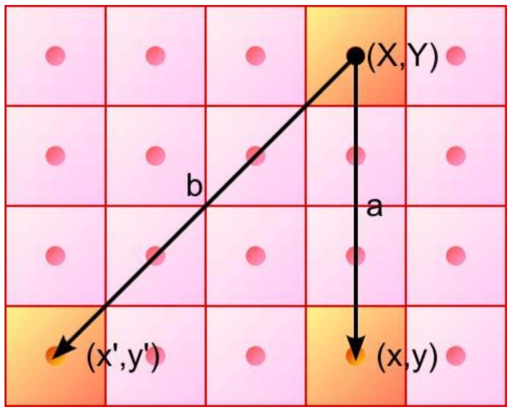


Hence, rays from the source point to all boundary pixels and cover all pixels of the image.

The number of algorithm iterations is `2*(Height+Width)*M`, where `Height` and `Width` are the image height and width, in pixels, and `M` is the maximum possible ray length: `M = max (Height, Width)`. The computational complexity of the algorithm is $O(M^2)$, which is significantly less than in the original algorithm.

Features of the Optimized Algorithm

The Decay value determines the multiplier before every summand in the sum. However, the step length depends on the angle between the ray and sample edges. For example, the source pixel (X,Y) contribution is the same for the pixel (x,y) and for the pixel (x’,y’), as shown in the figure below:



As a result, the God Rays effect is square shaped. To correct this artifact, adjust the Decay value to the algorithm step length:

1. Restore Decay as $e^{-\beta s}$, where s is the distance between pixels and β is the extinction constant composed of light absorption and out-scattering properties.
2. Compute *segment length* as s divided by the step count.

The original algorithm contains a fixed number of summands in a sum, so that convergence of a series does not affect the result. The result only changes when the step count N is changed. For example, the God Rays brightness increases when N is incremented.

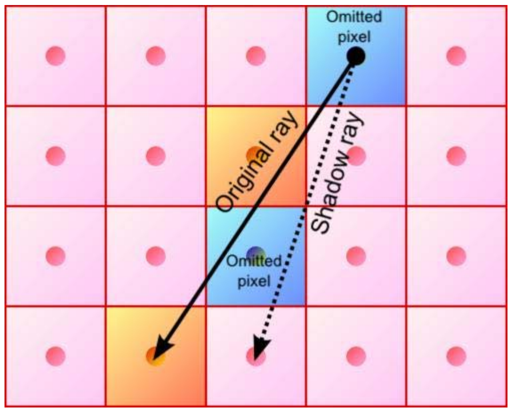
In the optimized algorithm, the number of summands varies from one in the God Rays source position to the maximum ray length. If Decay is less than one, the series tends to the value 1/(1-Decay), while the sum grows and then oscillates around its upper bound. This results in shading in the light source area. To correct this artifact, adjust the convergence of series to one by multiplying every summand by (1-Decay), except the last one. The last summand enables the sum to grow to the maximum value at the first step.

$$GodRaysColor_i = weight * (1 - decay) * PixelColor(x_i, y_i) + decay * GodRaysColor_{i-1}$$
$$GodRaysColor_0 = weight * PixelColor(x_0, y_0)$$

If the light source is beyond the image edges, you need to cut off the invisible parts of rays correctly. If you replace a cut-off ray with the ray starting in the center of the pixel crossed by the cut-off ray, the statement that all image pixels are filled in is not true, and missing pixels can appear.

Enhancing the Optimized Algorithm

You can improve the algorithm by parallelizing the workflow and calculating every ray in a separate thread. However, more than one ray can cross one pixel. This means that you can compute more than one value for a pixel. Practically, choosing any of the values is not a critical error, although the error can accumulate if you choose the value for all rays. A workaround of computing all values leads to multiple writes in the same memory area. To avoid this artifact, while calculating coordinates of the current ray, calculate the coordinates of corresponded steps of the subsequent ray in one thread. Use this additional ray to find pixels to be omitted. If the current pixel coordinates of these two rays match, the current ray pixel is omitted in the original ray and is filled by the subsequent ray in another thread. The additional ray (shadow ray) "shades" some pixels in the original ray. Therefore, the shadow ray is the ray to the border pixel next to the destination pixel of the original ray. The improved algorithm uses the anticlockwise direction, as shown in the following figure:



In explicit version, calculating several rays simultaneously is most efficient due to explicit usage of CPU SIMD units. For those original rays, the sample needs to calculate shadow rays at the same time. Starting from the second ray in a bunch, each original ray is a shadow ray for the previous one. Actually, the sample needs only the shadow ray for the last 15 rays. The implementation calculates coordinates of 16 rays in a row instead of 30 rays. If you cut off certain rays by image edges, you need to precisely calculate initial and final algorithm iterations.

Addition to the Optimized Algorithm

Use the filtering by depth buffer values to avoid bogus God Rays from highlighted parts of the foreground scene. If pixels are placed closer to the image plane, they cannot produce God Rays.

Please refer to the improved God Rays algorithm in the `EvaluateRay` function implementation in `GodRays.cl`.

OpenCL™ Implementation

This sample applies the following stages of the modified God Rays Pass simulation algorithm to an HDR image:

- application of God Rays to highlights in the input frame
- storing the result of the applied algorithm in the intermediate buffer called the God Rays mask

The current sample implementation rearranges the algorithm kernel to optimize it for the underlying CPU.

Code Highlights

The `GodRays` OpenCL kernel of the `GodRays.cl` file performs the God Rays effect. Every input bunch of rays has a unique global ID that the kernel uses for their identification. They are processed by OpenCL kernel function `EvaluateRay` called from `GodRays`. The God Rays effect sequence consists of OpenCL kernel call performed in `ExecuteGodRaysKernel()` function of `GodRays.cpp` file.

Work-group Size Considerations

You can specify any work-group size for this kernel. However, the kernel achieves peak performance for 1600x1200 two-dimensional HDR image with work-group size ranging from 1 to 16 elements.

Understanding OpenCL Performance Characteristics

Benefits of Using Vector Data Types

This sample implements the God Rays effect algorithm using vector data types. Explicit usage of vector types, such as `float4`, enables the following CPU optimizations:

- You can work with quadruples instead of single floats. This removes unnecessary branches, saves memory bandwidth, and optimizes CPU cache usage.
- You can use God Rays effect for a single four-color channels pixel item. Consequently, you can perform God Rays effect for four-color channels of an image pixel (RGBA pixel) and for four monochrome pixels simultaneously. The current version uses vector `float4` data types. As a result, you can achieve ~3x speedup for the current version of kernel as compared to the scalar version of the kernel.

Limitations

The current implementation produces natural-looking results mostly in the background, for example, God Rays in the sky. In the foreground, some God Rays artifacts can occur. To remove these artifacts, such as bogus God Rays, you need depth information.

Future Work and Enhancements

The sample performs all calculations in floating-point values. Each image pixel consists of four 32-bit floating-point values representing red, green, blue, and alpha (RGBA) image channels. You can improve this sample performance by introducing the following:

- more compact data representation
- additional Gaussian blurring (smoothing)
- tone mapping, for example, OpenEXR algorithm [2]

- auto-adjusting or tone-mapping parameters for the whole image/frame (auto-exposure)
- replacing a global tone mapping operator with a local tone mapping operator to adjust its parameters according to the local lighting conditions on the image/frame
- application of sophisticated depth control algorithms to produce more natural-looking God Rays in the foreground.

Project Structure

This sample project has the following structure:

- `GodRays.cpp` - the host code, with OpenCL initialization and processing functions
- `GodRays.cl` – source code of the OpenCL God Rays kernel
- `GodRaysNative.cpp` **–** source code of the native God Rays kernel implementation (SIMD)
- `GodRays.vcproj` **–** Microsoft* Visual Studio* 2008 project file.

APIs Used

This sample uses the following APIs:

- `clKreateKernel`
- `clCreateContextFromType`
- `clGetContextInfo`
- `clCreateCommandQueue`
- `clCreateProgramWithSource`
- `clBuildProgram`
- `clCreateBuffer`
- `clSetKernelArg`
- `clEnqueueNDRangeKernel`
- `clEnqueueReadBuffer`
- `clReleaseMemObject`
- `clReleaseKernel`
- `clReleaseProgram`
- `clReleaseCommandQueue`
- `clReleaseContext.`

Reference (Native) Implementation

Reference implementation is done in `ExecuteGodRaysReference()` routine of `GodRays.cpp` file. This is single-threaded code that performs exactly the same God Rays effect sequence as the OpenCL implementation, but using conventional nested loop in C with SSE optimizations. Native kernel `EvaluateRay()` that processes ray bunches is located in `GodRaysNative.cpp`.

Controlling the Sample

The sample executable is a console application. The current implementation has no command line arguments.

References

[1] Kenny Mitchell. GPU Gems 3, *Volumetric Light Scattering as a Post-Process.*

[2] http://www.openexr.com

[3] http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

[4] http://xw2k.nist.gov/dads/HTML/bresenham.html